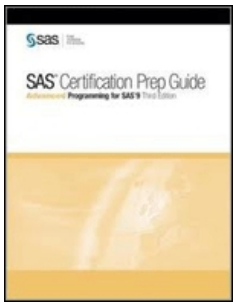


SAS Certification Prep Guide: Advanced Programming for SAS 9, Third Edition

by SAS Institute

Chapter 3: Combining Tables Horizontally Using PROC SQL

Overview

Introduction

When you need to select data from multiple tables and combine the tables *horizontally* (side by side), PROC SQL can be an efficient alternative to other SAS procedures or the DATA step. You can use a PROC SQL *join* to combine tables horizontally:

```
proc sql;
   select *
      from a, b
      where a.x=b.x;
```

Table A	Table B

A PROC SQL join is a query that specifies multiple tables and/or views to be combined and, typically, specifies the conditions on which rows are matched and returned in the result set.

You should already be familiar with the basics of using PROC SQL to join tables. In this chapter, you will take a more in-depth look at joining tables.

Objectives

In this chapter, you learn to

- combine tables horizontally to produce a Cartesian product
- combine tables horizontally using inner and outer joins
- use a table alias in a PROC SQL query
- use an in-line view in a PROC SQL query
- compare PROC SQL joins with DATA step match-merges and other SAS techniques
- use the COALESCE function to overlay columns in PROC SQL joins.

Prerequisites

Before you begin this chapter, you should complete the following chapters:

- "Performing Queries Using PROC SQL" on page 4
- "Performing Advanced Queries Using PROC SQL" on page 29.

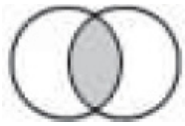
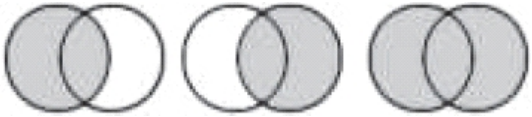
Understanding Joins

Joins combine tables horizontally (side by side) by combining rows. The tables being joined are not required to have the same number of rows or columns.

Note You can use a join to combine views as well as tables. Most of the following references to tables are also applicable to views; any exceptions are noted. In-line views are introduced later in this chapter. For more information about PROC SQL views, see "Creating and Managing Views Using PROC SQL" on page 260.

When you use a PROC SQL query to join tables, you must decide how you want the rows from the various tables to be combined. There are two main types of joins, as shown below.

Type of Join	Output

Inner join 	Only the rows that *match* across all table(s)
Outer join 	Rows that *match* across tables (as in the inner join) plus *nonmatching* rows from one or more tables

When any type of join is processed, PROC SQL starts by generating a *Cartesian product*, which contains all possible combinations of rows from all tables, consider how a Cartesian product is generated.

Generating a Cartesian Product

The most basic type of join combines data from two tables that are specified in the FROM clause of a SELECT statement. When you specify multiple tables in the FROM clause but do not include a WHERE statement to subset data, PROC SQL returns the *Cartesian product* of the tables. In a Cartesian product, each row in the first table is combined with every row in the second table. Below is an example of this type of query, which joins the tables *One* and *Two*.

```
proc sql;
   select *
      from one, two;
```

One

X	A
1	a
2	b
4	d

Two

X	B
2	x
3	y
5	v

X	A	X	B
1	a	2	x
1	a	3	y
1	a	5	v
2	b	2	x
2	b	3	y
2	b	5	v
4	d	2	x
4	d	3	y
4	d	5	v

The output shown above displays all possible combinations of each row in table *One* with all rows in table *Two*. Note that each table has a column named **X**, and both of these columns appear in the output. A Cartesian product includes all columns from the source tables; columns that have common names are *not* overlaid.

In most cases, generating all possible combinations of rows from multiple tables does *not* yield useful results, so a Cartesian product is rarely the query outcome that you want. For example, in the Cartesian product of two tables that

contain employee information, each row of output might contain information about two different employees. Usually, you want your join to return only a subset of rows from the tables.

The size of a Cartesian product can also be problematic. The number of rows in a Cartesian product is equal to the product of the number of rows in the contributing tables.

The tables *One* and *Two*, used in the preceding example, contain three rows each, as shown below.

One

X	A
1	a
2	b
4	d

Two

X	B
2	x
3	y
5	v

The number of rows in the Cartesian product of tables *One* and *Two* is calculated as follows:

```
3 x 3 = 9 rows
```

Joining small tables such as *One* and *Two* results in a relatively small Cartesian product. However, the Cartesian product of large tables can be huge and can require a large amount of system resources for processing.

For example, joining two tables of 1,000 rows each results in output of the following size:

```
1,000 x 1,000 = 1,000,000 rows
```

When you run a query that involves a Cartesian product that cannot be optimized, PROC SQL writes the following warning message to the SAS log.

Table 3.1: SAS Log

```
NOTE: The execution of this query involves performing one or more Cartesian product joins that
cannot be optimized.
```

Although you will not often choose to create a query that returns a Cartesian product, it is important to understand how a Cartesian product is built. In all types of joins, PROC SQL generates a Cartesian product first, and then eliminates rows that do not meet any subsetting criteria that you have specified.

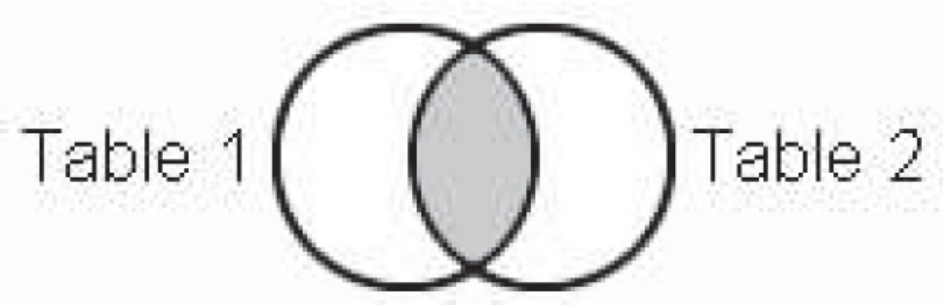
Note In many cases, PROC SQL can optimize the processing of a join, thereby minimizing the resources that are required to generate a Cartesian product.

Using Inner Joins

Introducing Inner Join Syntax

An inner join combines and displays only the rows from the first table that match rows from the second table, based on the matching criteria (also known as join conditions) that are specified in the WHERE clause. A join condition is an expression that specifies the column(s) on which the tables are to be joined.

The following diagram illustrates an inner join of two tables. The shaded area of overlap represents the matching rows (the subset of rows) that the inner join returns as output.



Note An inner join is sometimes called a *conventional join*.

Inner join syntax builds on the syntax of the simplest type of join that was shown earlier. In an inner join, a WHERE clause is added to restrict the rows of the Cartesian product that will be displayed in output.

General form, SELECT statement for inner join:

```
SELECT column-1<,…column-n>
                FROM table-1 | view-1, table-2 | view-2<,…table-n | view-n>
                WHERE join-condition(s)

                          <AND other subsetting condition(s)>
             <other clauses>;
```

where

join-condition(s)

refers to one or more expressions that specify the column or columns on which the tables are to be joined.

other subsetting condition(s)

refers to optional expressions that are used to subset rows in the query results.

<other clauses>

refers to optional PROC SQL clauses.

Note The maximum number of tables that you can combine in a single inner join depends on your version of SAS. For more information see the SAS documentation. If the join involves views (either in-line views or PROC SQL views), it is the number of tables that underlie the views, not the number of views themselves, that counts toward the limit. In-line views are covered later in this chapter and PROC SQL views are discussed in "Creating and Managing Views Using PROC SQL" on page 260.

Example

When a WHERE clause is added to the PROC SQL query shown earlier, only a subset of rows is included in output. The modified query, tables, and output are shown below:

```
proc sql;
   select *
      from one, two
      where one.x = two.x;
```

One

X	A
1	a
2	b
4	d

Two

X	B
2	x
3	y
5	v

X	A	X	B
2	b	2	x

Because of the WHERE clause, this inner join does *not* display *all* rows from the Cartesian product (all possible combinations of rows from both tables) but only a *subset* of rows. The WHERE clause expression (join condition) specifies that the result set should include only rows whose values of column **X** in the table *One* are *equal to* values in column **X** of the table *Two*. Only one row from *One* and one row from *Two* have matching values of **X**. Those two rows are combined into one row of output.

Note PROC SQL will not perform a join unless the columns that are compared in the join condition (in this example, **One.X** and **Two.X)** have the same data type. However, the two columns are *not* required to have the same name. For example, the join condition shown in the following WHERE statement is valid if **ID** and **EmpID** have the same data type:

```
where tablel.id = table2.empid
```

Note The join condition that is specified in the WHERE clause often contains the equal (=) operator, but the expression might contain one or more other operators instead. An inner join that matches rows based on the equal (=) operator, in which the value of a column or expression in one table must be equivalent to the value of a column or expression in another table, is called an *equijoin*.

Consider how PROC SQL processes this inner join.

Understanding How Joins Are Processed

Understanding how PROC SQL processes inner and outer joins will help you to understand which output is generated by each type of join. Conceptually, PROC SQL follows these steps to process a join:

- builds a Cartesian product of rows from the indicated tables
- evaluates each row in the Cartesian product, based on the join conditions specified in the WHERE clause (along with any other subsetting conditions), and removes any rows that do not meet the specified conditions
- if summary functions are specified, summarizes the applicable rows
- returns the rows that are to be displayed in output.

Note The PROC SQL query optimizer follows a more complex process than the conceptual approach described here,

by breaking the Cartesian product into smaller pieces. For each query, the optimizer selects the most efficient processing method for the specific situation.

By default, PROC SQL Joins do *not* overlay columns with the same name. Instead, the output displays *all* columns that have the same name. To avoid having columns with the same name in the output from an inner or outer join, you can eliminate or rename the duplicate columns.

Tip You can also use the COALESCE function with an inner or outer join to overlay columns with the same name. The COALESCE function is discussed, along with outer joins, later in this chapter.

Eliminating Duplicate Columns

Consider the sample PROC SQL query that uses an inner join to combine the tables *One* and *Two :*

```
proc sql;
   select *
      from one, two
      where one.x = two.x;
```

One

X	A
1	a
2	b
4	d

Two

X	B
2	x
3	y
5	v

X	A	X	B
2	b	2	x

The two tables have a column with an identical name **(x)**. Because the SELECT clause in the query shown above contains an asterisk, the output displays *all* columns from *both* tables.

To eliminate a duplicate column, you can specify just one of the duplicate columns in the SELECT statement. The SELECT statement in the preceding PROC SQL query can be modified as follows:

```
proc sql;
   select one.x, a, b
      from one, two
      where one.x = two.x
```

Here, the SELECT clause specifies that only the column **x** from table *One* will be included in output. The output, which now displays only one column **x**, is shown below.

X	A	B
2	b	x

Note In an inner equijoin, like the one shown here, it does not matter which of the same-named columns is listed in the SELECT statement. The SELECT statement in this example could have specified **Two.X** instead of **One.X**.

Another way to eliminate the duplicate **X** column in the preceding example is shown below:

```
proc sql;
   select one.*, b
      from one, two
      where one.x = two.x;
```

By using the asterisk (*) to select *all* columns from table *One*, and only **B** from table *Two*, this query generates the same output as the preceding version.

Renaming a Column by Using a Column Alias

If you are working with several tables that have a column with a common name but slightly different data, you might want *both* columns to appear in output. To avoid the confusion of displaying two different columns with the same name, you can rename one of the duplicate columns by specifying a column alias in the SELECT statement. For example, you could modify the SELECT statement of the sample query as follows:

```
proc sql;
   select one.x as ID, two.x, a, b
      from one, two
      where one.x = two.x;
```

The output of the modified query is shown here.

ID	X	A	B
2	2	b	x

Now that the column **One.X** has been renamed to **ID**, the output clearly indicates that **ID** and **X** are two different columns.

Joining Tables That Have Rows with Matching Values

Consider what happens when you join two tables in which multiple rows have duplicate values of the column on which the tables are being joined. Each of the tables *Three* and *Four* has multiple rows that contain the value *2* for column **X**. The following PROC SQL inner join matches rows from the two tables based on the common column **X:**

```
proc sql;
   select *
      from three, four
      where three.x=four.x;
```

Three

X	A
1	a1
1	a2
2	b1
2	b2
4	d

Four

X	B
2	x1
2	x2
3	y
5	v

The output shows how this inner join handles the duplicate values of **x**.

X	A	X	B
2	b1	2	x1
2	b1	2	x2
2	b2	2	x1
2	b2	2	x2

All possible combinations of the duplicate rows are displayed. There are no matches on any other values of **x**, so no other rows are displayed in output.

Note A DATA step match-merge would output only two rows, because it processes data sequentially from top to bottom. Later in this chapter, there is a comparison of PROC SQL Joins and DATA step match-merges.

Specifying a Table Alias

To enable PROC SQL to distinguish between same-named columns from different tables, you use qualified column names. To create a qualified column name, you prefix the column name with its table name. For example, the following PROC SQL inner join contains several qualified column names (shown highlighted):

```
proc sql;
title 'Employee Names and Job Codes';
   select staffmaster.empid, lastname, firstname, jobcode
      from sasuser.staffmaster, sasuser.payrollmaster
      where staffmaster.empid=payrollmaster.empid;
```

It can be difficult to read PROC SQL code that contains lengthy qualified column names. In addition, typing (and retyping) long table names can be time-consuming. Fortunately, you can use a temporary, alternate name for any or all tables in any PROC SQL query. This temporary name, which is called a *table alias*, is specified after the table name in the FROM clause. The keyword AS is often used, although its use is optional.

The following modified PROC SQL query specifies table aliases in the FROM clause, and then uses the table aliases to qualify column names in the SELECT and WHERE clauses:

```
proc sql;
title 'Employee Names and Job Codes';
   select s.empid, lastname, firstname, jobcode
      from sasuser.staffmaster as s,
           sasuser.payrollmaster as p
      where s.empid=p.empid;
```

In this query, the optional keyword AS is used to define the table aliases in the FROM clause. The FROM clause would be equally valid with AS omitted, as shown below:

```
from sasuser.staffmaster s,
     sasuser.payrollmaster p
```

Note While using table aliases will help you to work more efficiently, the use of table aliases does not cause SAS to execute the query more quickly.

Table aliases are usually optional. However, there are two situations that require their use, as shown below.

Table aliases are required when…	Example
a table is joined to itself (called a self-join or reflexive join)	`from airline.staffmaster as s1,` `airline.staffmaster as s2`

you need to reference columns from same-named tables in different libraries	`from airline.flightdelays as af,`
	`work.flightdelays as wf`
	`where af.delay > wf.delay`

So far, you have seen relatively simple examples of inner joins. However, as in any other PROC SQL query, inner joins can include more advanced components, such as

- titles and footers
- functions and expressions in the SELECT clause
- multiple conditions in the WHERE clause
- an ORDER BY clause for sorting
- summary functions with grouping.

Here are a few examples of more complex inner joins.

Example: Complex PROC SQL Inner join

Suppose you want to display the names (first initial and last name), job codes, and ages of all company employees who live in New York. You also want the results to be sorted by job code and age.

The data that you need is stored in the two tables listed below.

Table	Relevant Columns
Sasuser.Staffmaster	**EmpID, LastName, FirstName, State**
Sasuser.payrollmaster	**EmpID, JobCode, DateOfBirth**

Of the three columns that you want to display, **JobCode** is the only column that already exists in the tables. The other two columns will need to be created from existing columns.

The PROC SQL query shown here uses an inner join to generate the output that you want:

```
proc sql outobs=15;
title 'New York Employees';
   select substr(firstname,1,1) || '. ' || lastname
          as Name,
          jobcode,
          int((today() - dateofbirth)/365.25)
          as Age
```

```
      from sasuser.payrollmaster as p,
           sasuser.staffmaster as s
      where p.empid =
            s.empid
            and state='NY'
      order by 2, 3;
```

New York Employees		
Name	**JobCode**	**Age**
R. LONG	BCK	41
T. BURNETTE	BCK	45
J. MARKS	BCK	46
N. JONES	BCK	46
R. VANDEUSEN	BCK	52
J. PEARSON	BCK	53
L. GORDON	BCK	53
C. PEARCE	FA1	40
D. WOOD	FA1	41
C. RICHARDS	FA1	43
L. JONES	FA1	45
R. MCDANIEL	FA1	45
A. PARKER	FA1	48
D. FIELDS	FA1	54
R. PATTERSON	FA2	43

The SELECT clause, shown below, specifies the new column **Name**, the existing column **JobCode**, and the new column **Age:**

```
select substr(firstname,1,1) || '. ' || lastname
       as Name,
       jobcode,
       int((today() - dateofbirth)/365.25)
       as Age
```

To create the two new columns, the SELECT clause uses functions and expressions as follows:

- To create **Name**, the SUBSTR function extracts the first initial from **FirstName**. Then the concatenation operator combines the first initial with a period, a space, and then the contents of the **LastName** column. Finally, the keyword AS names the new column.
- To calculate **Age**, the INT function returns the integer portion of the result of the calculation. In the expression that is used as an argument of the INT function, the employee's birthdate **(DateOfBirth)** is subtracted from today's date (returned by the TODAY function), and the difference is divided by the number of days in a year (365.25).

The WHERE clause contains two expressions linked by the logical operator AND:

```
where p.empid =
      s.empid
      and state='NY'
```

This query only outputs rows that have matching values of **EmpID** *and* rows in which the value of **State** is *NY*. You do not need to prefix the column name **State** with a table name, because **State** occurs in only one of the tables.

Example: PROC SQL Inner Join with Summary Functions

You can also summarize and group data in a PROC SQL Join. To illustrate, modify the previous PROC SQL inner join so that the output displays the following summarized columns for New York employees in each job code: number of

employees and average age. The modified query is shown below:

```
proc sql outobs=15;
title 'Avg Age of New York Employees';
   select jobcode,
          count(p.empid) as Employees,
          avg(int((today() - dateofbirth)/365.25))
          format=4.1 as AvgAge
      from sasuser.payrollmaster as p,
           sasuser.staffmaster as s
      where p.empid =
            s.empid
            and state='NY'
      group by jobcode
      order by jobcode;
```

To create two new columns, the SELECT clause uses summary functions as follows:

- To create **Employees**, the COUNT function is used with **p.EmpID (Payrollmaster.EmpID)** as its argument.
- To create **AvgAge**, the AVG function is used with an expression as its argument. As described in the previous example, the expression uses the INT function to calculate each employee's age.

The output of this modified query is shown below.

Avg Age of New York Employees

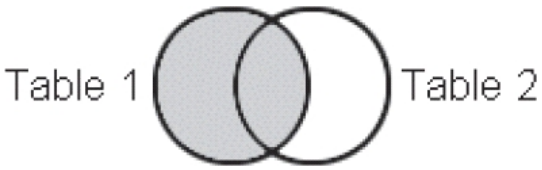
JobCode	Employees	AvgAge
BCK	7	48.0
FA1	7	45.1
FA2	9	48.7
FA3	4	50.8
ME1	5	44.0
ME2	9	52.1
ME3	2	52.0
NA1	1	42 0
NA2	1	52.0
PT1	5	43.8
PT2	5	58.0
PT3	2	65.0
SCP	6	48.5
TA1	5	47.6
TA2	12	47.1

Using Outer Joins

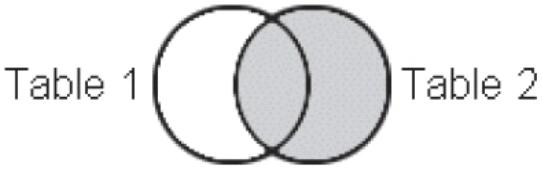
Introducing Types of Outer Joins

An outer join combines and displays all rows that *match* across tables, based on the specified matching criteria (also known as join conditions), plus some or all of the rows that do *not match*. You can think of an outer join as an augmentation of an inner join: an outer join returns all rows generated by an inner join, plus additional (nonmatching) rows.

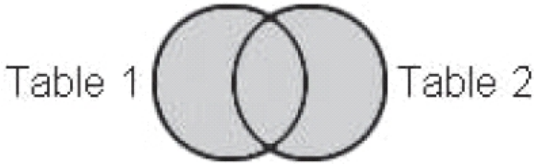
Type of Outer Join	Output
Left	All matching rows plus nonmatching rows from *the first* table specified in the FROM clause (the *left* table)



Right — All matching rows plus nonmatching rows from the *second* table specified in the FROM clause (the *right* table)



Full — All matching rows plus nonmatching rows in *both* tables



The syntax of an outer join is shown below.

General form, SELECT statement for outer join:

```
SELECT column-1<,…column-n>
     FROM table-1 | view-1
                  LEFT JOIN | RIGHT JOIN | FULL JOIN
                  table-2 | view-2
                  ON join-condition(s)
     <other clauses>;
```

where

LEFT JOIN, RIGHT JOIN, FULL JOIN

are keywords that specify the type of outer join.

ON

specifies *join-condition(s)*, which are expression(s) that specify the column or columns on which the tables are to be joined.

<other clauses>

refers to optional PROC SQL clauses.

Note To further subset the rows in the query output, you can follow the ON clause with a WHERE clause. The WHERE clause subsets the individual detail rows *before* the outer join is performed. The ON clause then specifies how the remaining rows are to be selected for output.

Note You can perform an outer join on only two tables or views at a time. Views are covered later in this chapter.

Consider how each type of outer join works.

Using a Left Outer Join

A left outer join retrieves *all rows that match across tables*, based on the specified matching criteria (join conditions), plus *nonmatching rows from the left table* (the *first* table specified in the FROM clause).

Suppose you are using the following PROC SQL left join to combine the two tables *One* and *Two*. The join condition is stated in the expression following the ON keyword. The two tables and the three rows of output are shown below:

```
proc sql;
   select *
      from one
       left join
       two
      on one.x=two.x;
```

One

X	A
1	a
2	b
4	d

Two

X	B
2	x
3	y
5	v

X	A	X	B
1	a	.	
2	b	2	x
4	d	.	

In each row of output, the first two columns correspond to table *One* (the *left* table) and the last two columns correspond to table *Two* (the *right* table).

Because this is a *left join*, all rows (both matching and nonmatching) from table *One* (the left table) are included in the output (the first two columns). Rows from table *Two* (the right table) are displayed in the output (the last two columns) only if they match a row from table *One*.

In this example, the second row of output is the only row in which the row from table *One* matched a row from table *Two*, based on the matching criteria (join conditions) specified in the ON clause. In the first and third rows of output, the row from table *One* had *no* matching row in table *Two*.

Note In all three types of outer joins (left, right, and full), the columns in the result (combined) row that are from the unmatched row are set to missing values.

To eliminate one of the duplicate columns (in this case, **x)** in any outer join, as shown earlier with an inner join, you can modify the SELECT clause to list the specific columns that will be displayed. Here, the SELECT clause from the preceding query has been modified to remove the duplicate **x** column:

```
proc sql;
    select one.x, a, b
       from one
       left join
       two
       on one.x=two.x;
```

One

X	A
1	a
2	b
4	d

Two

X	B
2	x
3	y
5	v

X	A	B
1	a	
2	b	x
4	d	

Using a Right Outer Join

A right outer join retrieves *all rows that match across tables*, based on the specified matching criteria (join conditions), plus *nonmatching rows from the right table* (the *second* table specified in the FROM clause).

Consider what happens when you use a right join to combine the two tables used in the previous example. The following PROC SQL query uses a right join to combine rows from *One* and *Two*, based on the join conditions specified in the ON clause:

```
proc sql;
   select *
      from one
      right join
      two
      on one.x=two.x;
```

One

X	A
1	a
2	b
4	d

Two

X	B
2	x
3	y
5	v

X	A	X	B
2	b	2	x
.		3	y
.		5	v

In each row of output, the first two columns correspond to table *One* (the left table) and the last two columns correspond to table *Two* (the right table).

Because this is a *right join*, all rows (both matching and nonmatching) from table *Two* (the right table) are included in the output (the last two columns). Rows from table *One* (the left table) are displayed in the output (the first two columns) only if they match a row from table *Two*.

In this example, there is only one row in table *One* that matches a value of **x** in table *Two*, and these two matching rows combine to form the first row of output. In the remaining rows of output, there is no match and the columns corresponding to table *One* are set to missing values.

Using a Full Outer Join

A full outer join retrieves *both matching rows and nonmatching rows* from both tables.

Combine the same two tables again, this time using a full join. The PROC SQL query, the tables, and the output are shown below:

```
proc sql;
   select *
      from one
      full join
      two
      on one.x=two.x;
```

One

X	A
1	a
2	b
4	d

Two

X	B
2	x
3	y
5	v

X	A	X	B
1	a	.	
2	b	2	x
.		3	y
4	d	.	
.		5	v

Because this is a *full join*, all rows (both *matching and nonmatching)* from *both* tables are included in the output. There is only one match between table *One* and table *Two*, so only one row of output displays values in all columns. All remaining rows of output contain only values from table *One* or table *Two*, with the remaining columns set to missing values.

Example: Outer Join

Now that you have seen how the three types of outer joins work, consider a realistic situation requiring the use of an outer join.

Suppose you want to list all of an airline's flights that were scheduled for March, along with corresponding delay information (if it exists). Each flight is identified by both a flight date and a flight number. Your output should display the following data: flight date, flight number, destination, and length of delay in minutes.

The data that you need is stored in the two tables shown below. The applicable columns from each table are identified.

Table	Relevant Columns
Sasuser.Marchflights	**Date, FlightNumber Destination**
Sasuser.Flightdelays	**Date, FlightNumber Destination, Delay**

Your output should include the columns that are listed above and all of the following rows:

- rows that have matching values of Date and FlightNumber across the two tables
- rows from *Sasuser.Marchflights* that have no matching row in *Sasuser.Flightdelays*.

To generate the output that you want, the following PROC SQL query uses a *left outer join*, with *Sasuser.Marchflights* specified as the *left* (first) table.

```
proc sql outobs=20;
title 'All March Flights';
   select m.date,
          m.flightnumber
                 label='Flight Number',
          m.destination
          label='Left',
```

```
    f.destination
    label='Right',
    delay
    label='Delay in Minutes'
from sasuser.marchflights as m
     left join
     sasuser.flightdelays as f
on m.date=f.date
   and m.flightnumber=
       f.flightnumber
order by delay;
```

Notice the following:

- The SELECT clause eliminates the duplicate **`Date`** and **`FlightNumber`** columns by specifying their source as *Sasuser.Marchflights*. However, the SELECT clause list specifies the **`Destination`** columns from both tables and assigns a table alias to each to distinguish between them.
- The ON clause contains two join conditions, which match the tables on the two columns **`Date`** and **`FlightNumber`**.

The query output is shown below.

All March Flights

Date	Flight Number	Left	Right	Delay in Minutes
14MAR2000	271	CDG		.
16MAR2000	622	FRA		.
.	132	YYZ		.
22MAR2000	183	WAS		.
11MAR2000	290	WAS		.
27MAR2000	982	DFW		.
29MAR2000	829	WAS		.
11MAR2000	202	ORD		.
08MAR2000	182	YYZ		.
17MAR2000	182	YYZ		.
03MAR2000	416	WAS		.
25MAR2000	872	LAX		.
09MAR2000	821	LHR	LHR	-10
25MAR2000	829	WAS	WAS	-10
02MAR2000	387	CPH	CPH	-10
10MAR2000	523	ORD	ORD	-10
07MAR2000	523	ORD	ORD	-10
18MAR2000	219	LHR	LHR	-10
14MAR2000	829	WAS	WAS	-10
27MAR2000	182	YYZ	YYZ	-9

The first 12 rows of output display rows from *Sasuser.Marchflights* (the left table) that have no matching rows in *Sasuser.Flightdelays*. Therefore, in these 12 rows, the last 2 columns are set to missing values.

Note The same results could be generated by using a *right* outer join, with *Sasuser.Marchflights* specified as the *right* (second) table.

Creating an Inner Join with Outer Join-Style Syntax

If you want to use a consistent syntax for all joins, you can write an inner join using the same style of syntax that is used for an outer join.

General form, SELECT statement for inner join (alternate syntax):

```
SELECT column-1<,…column-n>
     FROM table-1 | view-1
          INNER JOIN
          table-2 | view-2
          ON join-condition(s)
     <other clauses>;
```

where

INNER JOIN

is a keyword.

ON

specifies *join-condition(s)*, which are expression(s) that specify the column or columns on which the tables are to be joined.

<other clauses>

refers to optional PROC SQL clauses.

Note An inner join that uses this syntax can be performed on only two tables or views at a time. When an inner join uses the syntax that was presented earlier, up to 256 tables or views can be combined at once. In-line views are covered later in this chapter.

Comparing SQL Joins and DATA Step Match-Merges

Overview

You should be familiar with the use of the DATA step to merge data sets. DATA step match-merges and PROC SQL Joins can produce the same results. However, there are important differences between these two techniques. For example, a join does not require that you sort the data first; a DATA step match-merge requires that the data be sorted.

Compare the use of SQL joins and DATA step match-merges in the following situations:

- when *all* of the values of the selected variable (column) match
- when *only some* of the values of the selected variable (column) match.

When All of the Values Match

When *all* of the values of the BY variable match, you can use a *PROC SQL inner join* to produce the *same results* as a DATA step match-merge.

Suppose you want to combine the tables *One* and *Two*, as shown below.

One

X	A
1	a
2	b
3	c

Two

X	B
1	x
2	y
3	z

These two tables have the column **x** in common, and all values of **x** in each row match across the two tables. Both tables are already sorted by **x**.

The following DATA step match-merge (followed by a PROC PRINT step) and the PROC SQL inner join produce identical reports.

DATA Step Match-Merge

```
data merged;
   merge one two;
   by x; run;

  proc print data=merged noobs;
   title 'Table Merged';
run;
```

PROC SQL Inner Join

```
proc sql;
 title 'Table Merged';
    select one.x, a, b
       from one, two
       where one.x = two.x
       order by x;
```

One

X	A
1	a
2	b
3	c

Two

X	B
1	x
2	y
3	z

X	A	B
1	a	x
2	b	y
3	c	z

Note The DATA step match-merge creates a data set whereas the PROC SQL inner join, as shown here, creates only a report as output. To make these two programs completely identical, the PROC SQL inner join could be rewritten to create a table. For detailed information about creating tables with PROC SQL, see "Creating and Managing Tables Using PROC SQL" on page 175.

Note If the order of rows in the output does not matter, the ORDER BY clause can be removed from the PROC SQL join. Without the ORDER BY clause, this join is more efficient, because PROC SQL does not need to make a second pass through the data.

When Only Some of the Values Match

When *only some* of the values of the BY variable match, you can use a *PROC SQL full outer join* to produce the same result as a DATA step match-merge. Unlike the DATA step match-merge, however, a PROC SQL outer join does *not overlay* the two common columns by default. To overlay common columns, you must use the COALESCE function in the PROC SQL full outer join.

Note The COALESCE function can also be used with inner join operators.

Consider what happens when you use a PROC SQL full outer join *without* the COALESCE function. Suppose you want to combine the tables *Three* and *Four*. These two tables have the column **x** in common, but most of the values of **x** do *not* match across tables. Both tables are already sorted by **x**. The following DATA step match-merge (followed by a PROC PRINT step) and the PROC SQL outer join combine these tables, but do *not* generate the same output. The COALESCE function can also be used with inner join operators.

DATA Step Match-Merge

```
data merged;
   merge three four;
   by x;
run;

proc print data=merged noobs;
   title 'Table Merged';
run;
```

PROC SQL Full Outer Join

```
proc sql;
title 'Table Merged';
    select three.x, a, b
       from three
       full join
       four
       on three.x = four.x
       order by x;
```

Three

X	A
1	a
2	b
4	d

Four

X	B
2	x
3	y
5	v

DATA Step Match-Merge Output

Table Merged

X	A	B
1	a	
2	b	x
3		y
4	d	
5		v

PROC SQL Full Outer Join Output

Table Merged

X	A	B
.		y
.		v
1	a	
2	b	x
4	d	

The DATA step match-merge automatically overlays the common column, **x**. The PROC SQL outer join selects the value of **x** from just one of the tables, table *Three*, so that no **x** values from table *Four* are included in the PROC SQL output. However, the PROC SQL outer join cannot overlay the columns by default. The values that vary across the two merged tables are in bold above.

Consider how the COALESCE function is used in the PROC SQL outer join to overlay the common columns.

When Only Some of the Values Match: Using the COALESCE Function

When you add the COALESCE function to the SELECT clause of the PROC SQL outer join, the PROC SQL outer join can produce the same result as a DATA step match-merge.

General form, COALESCE function in a basic SELECT clause:

SELECT COALESCE *(column-1<,…column-n>)*

where

column-1 through *column-n*

are the names of two or more columns to be overlaid. The COALESCE function requires that all arguments have the same data type.

The COALESCE function overlays the specified columns by

- checking the value of each column in the order in which the columns are listed
- returning the first value that is a SAS nonmissing value.

Note If all returned values are missing, COALESCE returns a missing value.

When the COALESCE function is added to the preceding PROC SQL full outer join, the DATA step match-merge (with PROC PRINT step) and the PROC SQL full outer join will combine rows in the *same* way. The two programs, the tables, and the output are shown below.

DATA Step Match-Merge	PROC SQL Full Outer Join

`data merged;`
`  merge three four;`
`  by x;`
`run;`
`proc print data=merged noobs;`
`  title 'Table Merged';`
`run;`
	`proc sql;`
	`title 'Table Merged';`
	`  select coalesce(three.x, four.x)`
	`  as X, a, b`
	`  from three`
	`  full join`
	`  four`
	`  on three.x = four.x;`

Three

X	A
1	a
2	b
4	d

Four

X	B
2	x
3	y
5	v

Table Merged

X	A	B
1	a	
2	b	x
3		y
4	d	
5		v

Understanding the Advantages of PROC SQL Joins

DATA step match-merges and PROC SQL joins both have advantages and disadvantages. Here are some of the main advantages of PROC SQL joins.

Advantage	Example

PROC SQL Joins do *not* require *sorted or indexed* tables.	`proc sql;`
	`  select table1.x, a, b`
	`  from table1`
	`  full join`
	`  table2`
	`  on table1.x = table2.x;`
	
	where *table-1* is *sorted* by column **X** and *table-2* is not

PROC SQL Joins do *not* require that the columns in join expressions have the *same name*.	`proc sql;`
	`  select table1.x, lastname,`
	`  status`
	`  from table1, table2`
	`  where table1.id =`
	`  table2.custnum;`

PROC SQL Joins can use comparison operators *other than the equal sign (=)*.	`proc sql;`
	`  select a.itemnumber, cost,`
	`  price`

```
from table1 as a,
     table2 as b
where a.itemnumber = b.itemnumber
      and a.cost>b.price;
```

Note Join performance can be substantially improved when the tables are indexed on the column(s) on which the tables are being joined. You can learn more about indexing in "Creating and Managing Indexes Using PROC SQL" on page 238.

You can learn more about the comparative advantages and disadvantages of DATA step match-merges and PROC SQL Joins in "Combining Data Horizontally" on page 534.

Using In-Line Views

Introducing In-Line Views

Sometimes, you might want to specify an *in-line view* rather than a table as the source of data for a PROC SQL query. An in-line view is a nested query that is specified in the outer query's *FROM clause*. (You should already be familiar with a subquery, which is a nested query that is specified in a WHERE clause.) An in-line view selects data from one or more tables in order to produce a temporary (or virtual) table that the outer query then uses to select data for output.

For example, the following FROM clause specifies an in-line view:

```
from select flightnumber, date,
            boarded/passengercapacity*100
            as pctfull
            format=4.1 label='Percent Full'
      from sasuser.marchflights)
```

This in-line view selects two existing columns **(`FlightNumber` and `Date`)** and defines the new column **`PctFull`** based on the table *Sasuser.Marchflights*.

Unlike a table, an in-line view exists only during query execution. Because it is temporary, an in-line view can be referenced only in the query in which it is defined. In addition, an in-line view can be assigned an alias but it cannot be assigned a permanent name.

Note In a FROM clause, you can also specify a PROC SQL view, which is a query that has been created (using the CREATE statement) and stored. You can learn more about creating PROC SQL views in "Creating and Managing Views Using PROC SQL" on page 260.

Note Unlike other queries, an in-line view cannot contain an ORDER BY clause.

There are two potential advantages to using an in-line view instead of a table in a PROC SQL query:

- The complexity of the code is usually reduced, so that the code is easier to write and understand.
- In some cases, PROC SQL might be able to process the code more efficiently.

Referencing an In-Line View with Other Views or Tables

The preceding FROM clause is from a simple PROC SQL query that references just one data source: the in-line view. However, a PROC SQL query can join multiple tables and in-line views. For example, the FROM clause shown below specifies both a table *(Sasuser.Flightschedule)* and an in-line view.

```
from sasuser.flightschedule,
     (select flightnumber, date,
            boarded/passengercapacity*100
            as pctfull
            format=4.1 label='Percent Full'
        from sasuser.marchflights)
```

Referencing Multiple Tables in an In-Line View

You can specify more than one table in the FROM clause of an in-line view, as shown in the following example:

```
from (select marchflights.flightnumber,
             marchflights.date,
             boarded/passengercapacity*100
             as pctfull
             format=4.1 label='Percent Full',
             delay
         from sasuser.marchflights,
              sasuser.flightdelays
         where marchflights.flightnumber=
               flightdelays.flightnumber
               and marchflights.date=
               flightdelays.date)
```

In other words, you can base an in-line view on a join.

Note Remember that each table that is referenced in an in-line view counts toward the 256-table limit for an inner join.

Assigning an Alias to an In-Line View

You can assign an alias to an in-line view just as you can to a table. In the following example, the alias **f** has been added in the first FROM clause to reference the table *Sasuser.Flightschedule* and the alias **m** is associated with the results from the in-line view. After the main FROM clause, a WHERE clause that uses both of the aliases has been added.

```
from sasuser.flightschedule as f,
     (select flightnumber, date
             boarded/passengercapacity*100
             as pctfull
             format=4.1 label='Percent Full'
         from sasuser.marchflights) as m
where m.flightnumber=f.flightnumber
      and m.date=f.date
```

Example: Query That Contains an In-Line View

Suppose you want to identify the air travel destinations that experienced the worst delays in March. You would like your output to show all of the following data:

- destination
- average delay
- maximum delay
- probability of delay.

Your PROC SQL query uses an in-line view to calculate all of the new columns except for the last one:

```
proc sql;
title "Flight Destinations and Delays";
   select destination,
          average format=3.0 label='Average Delay',
          max format=3.0 label='Maximum Delay',
          late/(late+early) as prob format=5.2
          label='Probability of Delay'
      from (select destination,
                   avg(delay) as average,
                   max(delay) as max,
                   sum(delay > 0) as late,
                   sum(delay <= 0) as early
                   from sasuser.flightdelays
                   group by destination)
      order by average;
```

Consider each clause of the outer query, starting with the FROM clause, because PROC SQL evaluates the FROM clause before the SELECT clause.

The *FROM clause* specifies an in-line view rather than a table. The in-line view (nested query) specifies the following columns that are in the table *Sasuser.Flightdelays* or are based on a column in that table:

- the existing column **Destination**
- the new column **Average**
- the new column **Max**
- thenewcolumn **Late**
- the new column **Early**.

The columns **Average, Max, Late**, and **Early** are all calculated by using summary functions.

In the calculation for the columns **Late** and **Early**, a Boolean expression is used as the argument for the summary function. A Boolean function resolves either to 1 (true) or 0 (false). For example, **Late** is calculated by taking the sum of the Boolean expression **delay > 0**. For every value of **Delay** that is greater than 0, the Boolean expression resolves to 1; values that are equal to or less than 0 resolve to 0. The SUM function adds all values of **Delay** to indicate the number of delays that occurred for each destination.

The in-line view concludes with the clause **group by destination**, specifying that the in-line view data should be grouped and summarized by the values of **Destination**.

If you submitted this in-line view (nested query) as a separate query, it would generate the following output.

Destination	average	max	late	early
CDG	9.192308	39	21	5
CPH	5.740741	26	16	11
DFW	2.721311	20	38	23
FRA	5.951538	34	14	12
LAX	4.666667	27	82	41
LHR	5 551724	30	39	19
ORD	3.032609	19	51	41
WAS	0.635762	15	76	75
YYZ	2.016667	14	36	24

Consider the outer query's SELECT and ORDER BY clauses:

```
proc sql;
title "Flight Destinations and Delays";
   select destination,
          average format=3.0 label='Average Delay',
          max format=3.0 label='Maximum Delay',
          late/(late+early) as prob format=5.2
          label='Probability of Delay'
      from (select destination,
                   avg(delay) as average,
                   max(delay) as max,
                   sum(delay > 0) as late,
                   sum(delay <= 0) as early
                   from sasuser.flightdelays
                   group by destination)
          order by average;
```

The outer query's *SELECT clause* specifies columns as follows:

- **Destination** is an existing column in the table.
- **Average** and **Max** are calculated in the in-line view, and are assigned labels and formats in this SELECT clause.
- **Prob** (with the label "Probability of Delay") is calculated in this SELECT clause by using two columns that were calculated in the in-line view: **Late** and **Early**. The outer query's SELECT clause can refer to the calculated columns

Late and **Early** without using the keyword CALCULATED because PROC SQL evaluates the inline view (the outer query's FROM clause) first.

The outer query's last clause is an *ORDER BY clause*. The output will be sorted by the values of **Average**.

This PROC SQL query generates the following output.

Flight Destinations and Delays			
Destination	**Average Delay**	**Maximum Delay**	**Probability of Delay**
WAS	1	15	0.50
YYZ	2	14	0.60
DFW	3	20	0.62
ORD	3	19	0.55
LAX	5	27	0.67
LHR	6	30	0.67
CPH	6	26	0.59
FRA	6	34	0.54
CDG	9	39	0.81

Later in this chapter, you will see a PROC SQL query that combines multiple tables and uses an in-line view.

Joining Multiple Tables and Views

So far, this chapter has presented PROC SQL queries that combine only two tables horizontally. However, there might be situations in which you have to create complex queries to combine more than two tables. Here is an example of a complex query that combines four different tables.

Example: Complex Query That Combines Four Tables

Suppose you want to list the names of supervisors for the crew on the flight to Copenhagen on March 4, 2000. To solve this problem, you will need to use the following four tables.

Table	Relevant Columns
Sasuser.Flightschedule identifies the crew who flew to Copenhagen on March 4, 2000	**EmpID, Date, Destination**
Sasuser.Staffmaster identifies the names and states of residence for the employees	**EmpID, FirstName, LastName, State**
Sasuser.payrollmaster identifies the job categories for the employees	**EmpID, JobCode**
Sasuser.Supervisors identifies the employees who are supervisors	**EmpID, State, JobCategory**

Note Supervisors live in the same state as the employees they supervise. There is one supervisor for each state and job category.

This problem can be handled in a number of different ways. Examine and compare three different techniques:

- *Technique 1:* using PROC SQL subqueries, joins, and in-line views
- *Technique 2:* using a multi-way join that combines four different tables and a reflexive join (joining a table with itself)
- *Technique 3:* using traditional SAS programming (a series of PROC SORT and DATA steps, followed by a PROC PRINT step)

Example: Technique 1 (PROC SQL Subqueries, Joins, and In-Line Views)

Overview

Task **List the names of supervisors for the crew on the flight to Copenhagen on March 4, 2000.**

Data *Sasuser.Flightschedule* **(EmpID, Date, Destination)**

Sasuser.Staffmaster **(EmpID, FirstName, LastName, State)**

Sasuser.payrollmaster **(EmpID, JobCode)**

Sasuser.Supervisors **(EmpID, State, JobCategory)**

Note Supervisors live in the same state as the employees they supervise. There is one supervisor for each state and job category.

Completing the stated task requires a complex query that includes several subqueries, joins, and an in-line view. To make the task more manageable, build the complex query piece-by-piece in four steps:

1. Identify the *crew for the Copenhagen flight*.
2. Find the *states and job categories of the crew members* that were returned by the first query.
3. Find the *employee numbers of the crew supervisors*, based on the states and job categories that were returned by the second query.
4. Find the *names of the supervisors*, based on the employee numbers that were returned by the third query.

Note that at each of the four steps, a new piece of the final query will be added. The final query will include the four separate pieces.

Query 1: Identify the crew for the Copenhagen (CPH) flight

This query lists the employee ID numbers of all six crew members on the Copenhagen flight:

```
proc sql;
   select empid
      from sasuser.flightschedule
      where date='04mar2000'd
            and destination='CPH';
```

EmpID
1556
1830
1124
1135
1437
1839

Query 2: Find the states and job categories of the crew members

Query 1 becomes a subquery and returns the employee ID numbers of the six Copenhagen crew members to the outer query, *Query 2*. (Query 2 is shaded.) Query 2 uses an inner join to combine two tables. Query 2 selects the job category (by using the SUBSTR function to extract the job category from **JobCode)** and state for each of the six crew members.

```
proc sql;
   select substr(JobCode,1,2) as JobCategory,
          state
      from sasuser.staffmaster as s,
           sasuser.payrollmaster as p
      where s.empid=p.empid and s.empid in
         (select empid
            from sasuser.flightschedule
            where date='04mar2000'd
```

```
                  and destination='CPH');
```

JobCategory	State
FA	CT
FA	NY
NA	NY
PT	NY
PT	CT
FA	NY

Query 3: Find the employee numbers of the crew supervisors

Query 2 becomes an in-line view within *Query 3*, and the alias **c** has been assigned to the in-line view. Query 2 returns to Query 3 the job category and state for each crew member. Query 3 selects the employee ID numbers for supervisors whose job category and state match the job category and state of a crew member.

Note: Sasuser.Supervisors specifies the label **Supervisor ID** for the **EmpID** column, and this label appears in the output.

```
proc sql;
   select empid
      from sasuser.supervisors as m,
         (select substr(jobcode,1,2) as JobCategory,
                 state
            from sasuser.staffmaster as s,
                 sasuser.payrollmaster as p
            where s.empid=p.empid and s.empid in
               (select empid
                  from sasuser.flightschedule
                  where date='04mar2000'd
                        and destination='CPH')) as c
   where m.jobcategory=c.jobcategory
         and m.state=c.state;
```

Supervisor Id
1431
1983
1352
1118
1106
1983

Note that two rows contain the same value of **EmpID:** *1983*. This duplication indicates that two different crew members have the same manager. In all, there are five supervisors for the six crew members of the Copenhagen flight.

Query 4: Find the names of the supervisors

Query 3 becomes a subquery within Query 4. Query 3 returns to Query 4 the employee numbers (supervisor IDs) for the supervisors of the Copenhagen crew. Query 4 selects the names of the supervisors.

```
proc sql;
  select firstname, lastname
    from sasuser.staffmaster
    where empid in
      (select empid
```

```
      from sasuser.supervisors as m,
        (select substr(jobcode,1,2)
                as JobCategory,
                state
           from sasuser.staffmaster as s,
                sasuser.payrollmaster as p
           where s.empid=p.empid
                 and s.empid in
             (select empid
               from sasuser.flightschedule
               where date='04mar2000'd
                     and destination='CPH'))
                     as c
  where m.jobcategory=c.jobcategory
        and m.state=c.state);
```

FirstName	LastName
SHARON	DEAN
ROGER	DENNIS
JASPER	MARSHBURN
SIMON	RIVERS
DEBORAH	YOUNG

Note that the output has five rows, one for each supervisor. The duplicate name of a supervisor has been eliminated.

Technique 1 produces a PROC SQL query that includes

- four SELECT statements
- four tables, each read separately.

This program is not optimized and, in addition, includes complex code that is likely to take a long time to write.

Example: Technique 2 (PROC SQL Multi-way Join with Reflexive Join)

Task	**List the names of supervisors for the crew on the flight to Copenhagen on March 4, 2000.**
Data	*Sasuser.Flightschedule* **(EmpID, Date, Destination)**
	Sasuser.Staffmaster **(EmpID, FirstName, LastName, State)**
	Sasuser.payrollmaster **(EmpID, JobCode)**
	Sasuser.Supervisors **(EmpID, State, JobCategory)**

Note Supervisors live in the same state as the employees they supervise. There is one supervisor for each state and job category.

You can also solve this problem by using a multi-way join with a reflexive join (joining a table to itself). The code is shown below:

```
proc sql;
   select distinct e.firstname, e.lastname
      from sasuser.flightschedule as a,
           sasuser.staffmaster as b,
           sasuser.payrollmaster as c,
           sasuser.supervisors as d,
           sasuser.staffmaster as e
      where a.date='04mar2000'd and
            a.destination='CPH' and
            a.empid=b.empid and
            a.empid=c.empid and
            d.jobcategory=substr(c.jobcode,1,2)
```

```
            and d.state=b.state
            and d.empid=e.empid;
```

FirstName	LastName
DEBORAH	YOUNG
JASPER	MARSHBURN
ROGER	DENNIS
SHARON	DEAN
SIMON	RIVERS

Technique 2, which uses a multi-way join, provides a more efficient solution to the problem than Technique l. In a multi-way join, PROC SQL joins two tables at a time and performs the joins in the most efficient order (the order that will minimize the size of the Cartesian product). This multi-way join code is more difficult to build step-by-step than the code in Technique 1.

Note that *Sasuser.Staffmaster* is read two separate times in this query: this is the reflexive join. As you can see in the FROM clause, *Sasuser.Staffmaster* is assigned a different table alias each time it is read: first **b**, then **e**. The table is read the first time (alias **b**) to look up the states of the Copenhagen crew members, the second time (alias **e**) to look up the names of the supervisors.

Example: Technique 3 (Traditional SAS Programming)

Task	**List the names of supervisors for the crew on the flight to Copenhagen on March 4, 2000.**
Data	*Sasuser.Flightschedule* **(EmpID, Date, Destination)**
	Sasuser.Staffmaster **(EmpID, FirstName, LastName, State)**
	Sasuser.payrollmaster **(EmpID, JobCode)**
	Sasuser.Supervisors **(EmpID, State, JobCategory)**

Note Supervisors live in the same state as the employees they supervise. There is one supervisor for each state and job category.

For comparison, look at the traditional SAS programming that can be used to solve this problem. The code is shown below, followed by the output.

```
/* Find the crew for the flight. */

proc sort data=sasuser.flightschedule (drop=flightnumber)
          out=crew (keep=empid);
   where destination='CPH' and date='04MAR2000'd;
   by empid;
run;

/* Find the State and job code for the crew. */
proc sort data=sasuser.payrollmaster
               (keep=empid jobcode)
          out=payroll;
   by empid;
run;

proc sort data=sasuser.staffmaster
             (keep=empid state firstname lastname)
          out=staff;
   by empid;
run;

data st_cat (keep=state jobcategory);
   merge crew (in=c)
         staff
         payroll;
   by empid;
   if c;
```

```
   jobcategory=substr(jobcode,1,2);
run;

/* Find the supervisor IDs. */

proc sort
     data=st_cat;
        by jobcategory state;
run;

proc sort data=sasuser.supervisors
          out=superv;
   by jobcategory state;
run;

data super (keep=empid);
   merge st_cat(in=s)
         superv;
   by jobcategory state;
   if s;
run;

/* Find the names of the supervisors. */

proc sort data=super;
   by empid;
run;

data names(drop=empid);
   merge super (in=super)
         staff (keep=empid firstname lastname);
   by empid;
   if super;
run;

proc print data=names noobs uniform;
run;
```

LastName	FirstName
MARSHBURN	JASPER
DENNIS	ROGER
RIVERS	SIMON
YOUNG	DEBORAH
DEAN	SHARON
DEAN	SHARON

This output is *not* identical to the output of the PROC SQL approaches (Techniques 1 and 2). The SQL queries eliminated the duplicate names that are seen here. When you use Technique 3, you can eliminate duplicates by adding the NODUPKEY option to the last PROC SORT statement, as shown below:

```
proc sort data=super nodupkey;
```

Based on a mainframe benchmark in batch mode, the SQL queries use less CPU time, but more I/O operations, than this non-SQL program.

Summary

Contents

This section contains the following topics.

Text Summary

Understanding Joins

A PROC SQL Join is a query that combines tables horizontally (side by side) by combining rows. The two main types of joins are inner joins and outer joins.

Generating a Cartesian Product

When you specify multiple tables in the FROM clause but do not include a WHERE statement to subset data, PROC SQL returns the Cartesian product of the tables. In a Cartesian product, each row in the first table is combined with every row in the second table. In all types of joins, PROC SQL generates a Cartesian product first, and then eliminates rows that do not meet any subsetting criteria that you have specified.

Using Inner joins

An inner join combines and displays the rows from the first table that match rows from the second table, based on the matching criteria (also known as join conditions) that are specified in the WHERE clause. When the tables that are being joined contain a column with a common name, you might want to eliminate the duplicate column from results or specify a column alias to rename one of the duplicate columns. To refer to tables in an inner join, or in any PROC SQL step, you can specify a temporary name called a table alias.

Using Outer Joins

An outer join combines and displays all rows that match across tables, based on the specified matching criteria (also known as join conditions), plus some or all of the rows that do not match. There are three types of outer joins:

- A left outer join retrieves all rows that match across tables, based on the specified matching criteria (join conditions), plus nonmatching rows from the left table (the first table specified in the FROM clause).
- A right outer join retrieves all rows that match across tables, based on the specified matching criteria (join conditions), plus nonmatching rows from the right table (the second table specified in the FROM clause).
- A full outer join retrieves both matching rows and nonmatching rows from both tables.

Creating an Inner Join with Outer Join-Style Syntax

If you want to use a consistent syntax for all joins, you can write an inner join using the same style of syntax as used for an outer join.

Comparing SQL Joins and DATA Step Match-Merges

DATA step match-merges and PROC SQL joins can produce the same results, although there are important differences between these two techniques.

- When all the values of the BY variable (column) match and there are no duplicate BY variables, you can use a PROC SQL inner join.
- When only some of the values of the BY variable match, you can use a PROC SQL full outer join. To overlay common columns, you must use the COALESCE function with the PROC SQL Join.

Using In-Line Views

An in-line view is a subquery that appears in a FROM clause. An in-line view selects data from one or more tables to produce a temporary (or virtual) table that the outer query uses to select data for output. You can reference an in-line view with other views or tables, reference multiple tables in an in-line view, and assign an alias to an in-line view.

Joining Multiple Tables and Views

When you perform a complex query that combines more than two tables or views, you might need to choose between several different techniques.

Syntax

```
PROC SQL;
  SELECT column-1<,…column-n>
       FROM table-1 | view-1, table-2 | view-2 <,…table-n | view-n> | query-expression
       WHERE join-condition(s)
         <AND other subsetting condition(s)>
       <other clauses>;
  SELECT column-1<,…column-n>
       FROM table-1 | view-1
              LEFT JOIN | RIGHT JOIN | FULL JOIN
              table-2 | view-2
              ON join-condition(s)
       <other clauses>;
  SELECT column-1<,…column-n>
       FROM table-1 | view-1
              INNER JOIN
              table-2 | view-2
              ON join-condition(s)
       <other clauses>;
  SELECT COALESCE (column-1<,…column-n>)
       FROM table-1 | view-1
              LEFT JOIN | RIGHT JOIN | FULL JOIN
              table-2 | view-2
              ON join-condition(s)
       <other clauses>;
QUIT;
```

Sample Programs

Combining Tables by Using an Inner Join

```
proc sql outobs=15;
title 'New York Employees';
   select substr(firstname,1,1) || '. ' || lastname
          as Name,
          jobcode,
          int((today() - dateofbirth)/365.25)
          as Age
      from sasuser.payrollmaster as p,
           sasuser.staffmaster as s
      where p.empid =
            s.empid
            and state='NY'
      order by 2, 3;
quit;
```

Combining Tables by Using a Left Outer Join

```
proc sql outobs=20;
title 'All March Flights';
   select m.date,
          m.flightnumber
             label='Flight Number',
          m.destination
          label='Left',
          f.destination
          label='Right',
          delay
          label='Delay in Minutes'
      from sasuser.marchflights as m
           left join
           sasuser.flightdelays as f
      on m.date=f.date
         and m.flightnumber=
             f.flightnumber
      order by delay;
```

```
quit;
```

Overlaying Common Columns in a Full Outer Join

```
proc sql;
   select coalesce(p.empid, s.empid)
          as ID, firstname, lastname, gender
      from sasuser.payrollmaster as p
                full join
           sasuser.staffmaster as s
      on p.empid = s.empid
      order by id;
quit;
```

Joining Tables by Using a Subquery and an In-Line View

```
proc sql;
   select empid
      from sasuser.supervisors as m,
         (select substr(jobcode,1,2) as JobCategory,
                 state
            from sasuser.staffmaster as s,
                 sasuser.payrollmaster as p
            where s.empid=p.empid and s.empid in
               (select empid
                  from sasuser.flightschedule
                  where date='04mar2000'd
                        and destination='CPH')) as c
      where m.jobcategory=c.jobcategory
            and m.state=c.state;
quit;
```

Points to Remember

- In most cases, generating all possible combinations of rows from multiple tables does not yield useful results, so a Cartesian product is rarely the query outcome that you want.
- The maximum number of tables that you can combine in a single inner join depends on your version of SAS. If the join involves views, it is the number of tables that underlie the views, not the number of views, that counts towards the limit. An outer join can be performed on only two tables or views at a time.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. A Cartesian product is returned when ?

a. join conditions are *not* specified in a PROC SQL join.

b. join conditions are *not* specified in a PROC SQL set operation.

c. more than two tables are specified in a PROC SQL join.

d. the keyword ALL is used with the OUTER UNION operator.

2. Given the PROC SQL query and tables shown below, which output is generated? ?

```
proc sql;
   select *
      from store1,
      store2
      where store1.wk=
            store2.wk;
```

Store1

Wk	Sales
1	$515.07
2	$772.29
3	$888.88
4	$1000.01

Store2

Wk	Sales
1	$1368.99
2	$1506.23
3	$1200.57
4	$1784.11
5	$43.00

---

a.

Wk	Sales	Wk	Sales
1	$515.07	1	$1368.99
2	$772.29	2	$1506.23
3	$888.88	3	$1200.57
4	$1000.01	4	$1784.11
.	.	5	$43.00

b.

Wk	Sales	Wk	Sales
1	$515.07	1	$1368.99
2	$772.29	2	$1506.23
3	$888.88	3	$1200.57
4	$1000.01	4	$1784.11

c.

Wk	Sales
1	$515.07
2	$772.29
3	$888.88
4	$1000.01

d.

Wk	Sales
1	$515.07
2	$772.29
3	$888.88
4	$1000.01
1	$1368.99
2	$1506.23
3	$1200.57
4	$1784.11

3. Given the PROC SQL query and tables shown below, which output is generated? ?

```
proc sql;
   select s.*, bonus
      from bonus as b
      right join
      salary as s
      on b.id=
         s.id;
```

Bonus

ID	Bonus
123	5000
456	7000
744	3500

Salary

ID	Salary
123	70000
456	80000
978	55000

a.

Id	Salary	Bonus
123	70000	5000
456	80000	7000
978	55000	3500

b.

Id	Salary	Bonus
123	70000	5000
456	80000	7000
744	.	3500

c.

Id	Salary	Bonus
123	70000	5000
456	80000	7000
744	55000	3500

d.

Id	Salary	Bonus
123	70000	5000
456	80000	7000
978	55000	.

4. Which PROC SQL query produces the same output as the query shown here? ?

```
proc sql;
   select a.*,
          duration
```

```
    from groupa as a,
         groupb as b
    where a.obs=b.obs;
```

Note Assume that the table *Groupa* contains the columns **Obs** and **Med**. *Groupb* contains the columns **Obs** and **Duration**.

a. a. proc sql;
```
   select a.obs label='Obs',
          med
          b.obs label='Obs',
          duration
      from groupa as a, groupb as b
      where a.obs=b.obs;
```

b. b. proc sql;
```
   select coalesce(a.obs, b.obs)
          label='Obs', med, duration
      from groupa as a
      full join
      groupb as b
      on a.obs=b.obs;
```

c. c. proc sql;
```
   select a.*, duration
      from groupa as a
      left join
      groupb as b
      where a.obs=b.obs;
```

d. d. proc sql;
```
   select a.*, duration
      from groupa as a
      inner join
      groupb as b
      on a.obs=b.obs;
```

5. Which output will the following PROC SQL query generate? ?

```
proc sql;
   select *
      from table1
      left join
      table2
             on table1.g3=
                table2.g3;
```

Table1

G3	Z
89	FL
46	UI
47	BA

Table2

G3	R
46	BC
85	FL
99	BA

---

a.

G3	Z	G3	R
89	FL	.	
46	UI	46	BC
47	BA	.	

b.

G3	Z	G3	R
46	FL	46	BC
.		85	FL
.		99	BA

c.

G3	Z
46	UI

d.

G3	Z	G3	R
46	UI	46	BC

6. In order for PROC SQL to perform an inner join, ?

a. the tables being joined must contain the same number of columns.

b. the tables must be sorted before they are joined.

c. the columns that are specified in a join condition in the WHERE clause must have the same data type.

d. the columns that are specified in a join condition in the WHERE clause must have the same name.

7. Which statement about in-line views is false? ?

a. Once defined, an in-line view can be referenced in any PROC SQL query in the current SAS session.

b. An in-line view can be assigned a table alias but not a permanent name.

c. In-line views can be combined with tables in PROC SQL joins.

d. This PROC SQL query contains an in-line view that uses valid syntax:

```
proc sql;
   select name, numvisits
   from (select name, sum(checkin)
        as numvisits
           from facility as f, members as m
```

```
            where area='POOL' and
                  f.id=m.id
            group by name)
      where numvisits<=10
      order by 1;
```

8. Which PROC SQL query will generate the *same* output as the DATA step match-merge and PRINT step shown below? ?

```
data merged;
   merge table1 table2;
   by g3;
run;

proc print data=merged
     noobs;
   title 'Merged';
run;
```

Table1

G3	Z
46	UI
47	BA
89	FL

Table2

G3	R
46	BC
85	FL
99	BA

Merged

G3	Z	R
46	UI	BC
47	BA	
85		FL
89	FL	
99		BA

---

a. a.
```
proc sql;
title 'Merged';
 select a.g3, z, r
   from table1 as a
   full join
   table2 as b
   on a.g3 = b.g3
   order by 1;
```

b. b.
```
proc sql;
title 'Merged';
  select a.g3, z, r
    from table1 as a
    table2 as b
```

```
            on a.g3 = b.g3
            order by 1;
```

c.
```
c.    proc sql;
      title 'Merged';
        select coalesce(a.g3, b.g3)
            label='G3', z, r
        from table1 as a
        full join
        table2 as b
        on a.g3 = b.g3
        order by 1;
```

d.
```
d.   proc sql;
     title 'Merged';
       select g3, z, r
       from table1 as a
       full join
       table2 as b
       on a.g3 = b.g3
       order by 1;
```

9. In which of the following ways is a PROC SQL join different from a DATA step match-merge? ?

a. a PROC SQL join does not overlay common columns by default.

b. a PROC SQL join does not require the data to be sorted first.

c. a PROC SQL join produces printed output.

d. all of the above.

10. Which statement about the use of table aliases is false? ?

a. Table aliases must be used when referencing identical table names from different libraries.

b. Table aliases can be referenced by using the keyword AS.

c. Table aliases (or full table names) must be used when referencing a column name that is the same in two or more tables.

d. Table aliases must be used when using summary functions.

Answers

1. Correct answer: a

A Cartesian product is returned when join conditions are *not* specified in a PROC SQL join. In a Cartesian product, each row from the first table is combined with every row from the second table.

2. Correct answer: b

This PROC SQL query is an inner join. It combines the rows from the first table that match rows from the second table, based on the matching criteria specified in the WHERE clause. Columns are *not* overlaid, so *all* columns from the referenced tables (including any columns with duplicate names) are displayed. Any unmatched rows from either table are *not* displayed.

3. Correct answer: d

This PROC SQL query is a right outer join, which retrieves all rows that match across tables, based on the join conditions in the ON clause, plus nonmatching rows from the right (second)table.

4. Correct answer: d

There are two valid formats for writing a PROC SQL inner join. The PROC SQL query shown at the top of this question uses the first inner join format, which does *not* use a keyword to indicate the type of join. The alternate format is similar to an outer join and uses the keyword INNER JOIN.

5. Correct answer: a

This PROC SQL query is a left outer join, which retrieves all rows that match across tables (based on the join conditions in the ON clause), plus nonmatching rows from the left (first)table. No columns are overlaid, so all columns from both tables are displayed.

6. Correct answer: c

Inner Joins combine the rows from the first table that match rows from the second table, based on one or more join conditions in the WHERE clause. The columns being matched must have the same data type, but they are not required to have the same name. For joins, the tables being joined can have different numbers of columns, and the rows do not need to be sorted.

7. Correct answer: a

Unlike a table, an in-line view exists only during query execution. Because it is temporary, an in-line view can be referenced only in the query in which it is defined.

8. Correct answer: c

In order to generate the same output as the DATA step and PRINT steps, the PROC SQL full outer join must use the COALESCE function with the duplicate columns specified as arguments.

9. Correct answer: c

A maximum of 32 tables can be combined in a single inner join. If the join involves views (either in-line views or PROC SQL views), it is the number of tables that underlie the views, not the number of views, that counts towards the limit of 32.

10. Correct answer: d

The use of summary functions does *not* require the use of table aliases. All of the other statements about table aliases that are shown here are true.